University POLITEHNICA of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department

Diploma Thesis

Super Title

by

Nume Prenume

Supervisor: As. Drd. Ing. Andrei Voinescu

Bucharest, September 2014

Contents

Abstract

A User-friendly, Innovative Layout and Routing Algorithm for Diagrams

Diagrams are used in a variety of fields such as FPGA design as a means to represent workflows and concepts and require dedicated algorithms which can produce easy to understand results. Available software is clunky, at times it can be really slow and will often result in layouts which are hard to read and understand. The main issue of layout algorithms is the sheer amount of data that a diagram (graph) can contain, which makes placing and routing in a limited space extremely difficult. We attempted to solve this problem by using a two phase software which tries to generate layouts which feel natural for the user. The first phase of the software will layout the diagrams using a genetic algorithm and constraints provided by the user, while the second phase will route the paths connecting the diagrams using orthogonal routing. Using this approach, we can provide layouts which suite the needs and preferences of any user, instead of providing generic representations.

Keywords

Acknowledgements

Mulumiri

Chapter 1

Introduction

Diagrams are tools which are widely used in various domains in order to represent concepts, hierarchies, architectures etc. In software engineering, they can be found as an UML(Unified Modeling Language) visualisation of a system or program or as an inheritence tree in the case of OOP(Object Oriented Programming) languages. Diagrams are also present in the hardware world where their usage includes: logical representation of an FSM(Finite State Machine), data flow between components, on-chip dependencies between components. However, there is a field in which diagrams are a key part of development: hardware design and verification. In the case of hardware design, be it ASIC(Application Specific Integrated Circuit), FPGA(Field Programmable Gate Array), processors, memories or microcontrollers, diagrams are used to describe the individual blocks which are put together to form the respective chip, to represent the way modules are connected at a logical level or to show how data is passed and manipulated by each component. In hardware verification, they are used to represent hierarchies and inheritence amongst specific units such as agents, sequencers, test scoreboards and general test flows, often corresponding with the guidelines of accepted methodologies such as OVM(Open Verification Methodology) and UVM(Universal Verification Methodology). Therefore, one can assert that diagrams are instrumental in all stages of hardware design and verification.

Despite being an important part of the hardware design and verification world, diagrams do not receive the required attention from the developer community. While HDLs(Hardware Description Languages) have been gaining popularity over the years and IDEs have been developing to support them, diagrams have not received the adequate attention from the community. It is often overlooked that their main utility lies in understanding and maintaining the coherence and logical flow of a design or a verification testbench.

Usually, a hardware design consists of a number of design elements which can range from hundreds to thousands of design elements; the equivalent of a graph

with the same number of nodes. Representing such a design as a diagram would be, in turn, equivalent to drawing said graph in a plane. A given graph must pass through three steps in order to obtain a diagram representation: planarity testing, node placement and connection routing. For current algorithms and software, this is a well known problem which is difficult to solve in a reasonable amount of time and also yield a readable and easy to understand result. However, there are methods and algorithms which can be combined to obtain a feasible diagram representation.

The layout and routing solution which is presented in this study combines techniques used in current solutions with newer algorithms to in the attempt to obtain representations which respect the following criteria:

- Coherency: The diagram should be easy to understand and follow. Each connection in the diagram represents either data paths(BUS) or logical connections between components. For this reason the user must be able to understand where these paths start, where they end and what components they connect.
- Correctness: As any algorithm, the solution must maintain the correctness of the model when representing it. All nodes have to be included and all edges represented according to the corresponding adjacency matrix of the given graph.
- Planarity: Refers to the property of the given graph to be represented in the two dimensional plane. A planar graph does not have intersecting edges. It is expected that the representation preserves the planarity property of the graph.
- Orthogonality: Edges between nodes are represented as orthogonal connectors. Orthogonality implies that a path is composed only of perpendicular segments and contains a minimum number of bends. Paths shall not overlap any node.

Chapter 2

Related Work

Chapter 3

Algorithm Design

This chapter presents how the application is designed at a logical level and describes the main algorithms and techniques which are combined and used by the implementation.

3.1 Planarity testing

The first algorithm used by the application is the planarity testing method known as "path addition". In order to properly present this algorithm, we must first introduce the necessary notions regarding graph planarity and methods of testing this property.

3.1.1 Planarity property and criteria

A given graph G=(V,E) is planar if it can be drawn on a plane and its edges never cross each other, i.e. they intersect only at their endpoints. This type of drawing is also known as a planar embedding of the graph.

In order to determine if a graph possesses the planarity property, a series of theorems and criteria have been stated over the years. Amongst the first of these criteria is a theorem published by the Polish mathematician Kazimierz Kuratowski in 1930. The theorem deals with subdivisions, i.e. graphs which result from inserting vertices into edges. It states that a planar graph shall not contain a subdivision of the forbidden graphs K5(the complete graph on five vertices) or K3,3(complete bipartite graph on six vertices, three of which connect to each of the other three, also known as the utility graph). It is formulated as follows:

A finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K5 or K3,3 .

Another important theorem which deals with planarity was formulated by the German mathematician Klaus Wagner. This theorem takes into consideration graph

minors instead of subdivisions. A graph H is called a minor of a given graph G if H is obtained by deleting vertexes and edges or by contracting edges. The Wagner theorem states that a finite graph is planar if and only if it does not have K5 or K3,3 as a minor.

While these theorems manage to correctly define the planarity problem in a mathematical way, they are not optimal criterions to use in practice. The main reason is efficiency; we would like the complexity of such an algorithm to be linear O(n). In practice, there exist other theorems and criteria which fit in the linear complexity. For example, given a finite, connected planar graph with v the number of vertices, e the number of edges and f the number of faces (regions bounded by edges, including the outer, infinitely large region), the following hold true:

Theorem 1: If v 3 then e 3v 6; Theorem 2: If v 3 and there are no cycles of length 3, then e 2v 4; Theorem 3: v e + f = 2 (Euler's formula).

Unfortunately, these theorems are only necessary conditions, not sufficient conditions. They can only be used to prove that a graph is not planar; they cannot prove that a graph is planar.

3.1.2 Vertex addition method

The edge addition algorithm is the result an intensive research started in the 1960s by Abraham Lempel, Shimon Even and Cederbaum. They created an algorithm to determine whether a graph is planar or not and embed it in the plane in O(n2) time. Later on, Even and Tarjan developed a method to generate the st-numbering of a graph in linear O(n) time.

An st-numbering of a graph G is a subgraph H which has a set of properties, as follows. First, nodes in G are numbered. Then the vertex with the lowest number is designated as source vertex, *s*, and the one with the highest number is designated target, *t*. All the remaining vertexes which are not the source or target must be connected to a vertex numbered lower than them, and a vertex numbered higher than them. In order to achieve this configuration, the subgraph H may add additional nodes to the graph G.

Finally, Kellogg Booth and George Lueker created a data structure which represents the possible embeddings of a graph (or, in practice, induced subgraphs of a graph) called a PQ-tree. Using this data structure and the previous methods, they managed to created a linear time algorithm to determine planarity and possible embeddings. We will next present the general steps of an implementation of the Vertex addition method as proposed by Norishige Chiba and Takao Nishizekii.

First, it computes the st-numbering of the given graph. It then creates a PQ-tree which contains only one P node, the source, and all the other nodes are leaves. Next, a loop is entered which, for every leaf node, shall perform two steps:

****** itemizeu de mai jos o sa-l inlocuiescu cu un listing cu pseudocod *****

- A reduction step which attempts to gather all matching leaves into a P node which respects the st-numbering. If this step fails, then the graph is not planar.
- A vertex addition step, in which full nodes are replaced by a single P node and all the neighbours of that node numbered higher than itself are added as leaves.

  Figures x to y below present the steps of this algorithm applied on a graph G(3,3).

  ****** Insert figures here ******

3.2 Graph placement

Graph placement refers to assigning a position (set of coordinates) to each node of the graph in the space where the graph has to be represented.

We saw in the previous chapter that algorithms which determine if a graph is planar can also generate its embedings. However, this is not the same as actually drawing the graph. In reality, we are constrained by the limitations of the space in which we want to draw the diagram. Therefore, even though we know that a possible arrangement of the nodes will allow us to represent the graph in a plane, we cannot be certain that the representation is feasible or understandable. Furthermore, one may want to draw even graphs which are not planar, knowing that two or more of its edges will cross one another.

For this reason, a logical positioning of each node in the given plane is required. This step greatly facilitates the routing process and also helps reduce the time and complexity of said operation.

Commonly used methods which are also incorporated in the solution proposed by this thesis include grid placement and force directed graph-drawing. We shall continue by presenting these methods in the following sections.

3.2.1 Grid placement

Grid placement is one of the most popular and most used techniques by diagram drawing algorithms. It splits the space in which the representation is done into a grid and then starts placing the nodes in a certain order. Different approaches can be found here, depending on the type of the graph.

Trees are often placed in layeres. The root of the tree is first fixed on an edge of the grid, and then each level of the tree visited in breadth-first order forms a layer of nodes. Layers can be arranged vertically or horizontally, depending on the implementation.

More general graphs may also follow this pattern, by computing the spanning tree of the graph and placing nodes as stated above, starting with the spanning

tree's root. Other approaches for general graphs may place and pin the node with the highest number of connections in the middle of the grid.

The advantages of this method are represented by its ease of implementation, small consumption of resources and low complexity. A main drawback is that it generally interfers with the routing process and does not facilitate it. It also poses the risk of drawing planar graphs as though they were non-planar, i.e. intersecting various edges.

***** Aici poze cu grid-uri *****

3.2.2 Force directed graph-drawing

Force directed drawing is a technique which borrows concepts from physiques. In order to apply this method, one must model the graph as a physical system in which elements (nodes) interract with each other through forces. Connected nodes will be pulled towards each other by attraction forces (such as spring forces base on Hooke's Law), while unrelated nodes reject each other (similar to electrically charged particles according to Coulomb's Law). These forces are applied continuously until the system reaches stability.

Using such a technique will yield a graph placement which has the strongly connected nodes pulled in the middle of the assigned drawing spaced, while the individual or loosely connected nodes are pushed towards the edges. The same holds if the graph is split into multiple independet connected components: the nodes of each component shall be pulled towards each other, while the individual components will reject one another.

***** Poze cu mici galaxii dinastea ******

This approach is better suited to ease the routing process and reduce its time and complexity. It is less likely that planar graphs shall be improperly drawn and routes will be shorter and clearer. However, unlike Grid placement, it is an iterative process which may take considerable amounts of time to reach stability, while also consuming more resources.

3.3 Edge routing

The final step in drawing a diagram or a graph is represented by edge routing. Its purpose is to highlit the connections and dependencies between nodes via a set of lines which add up to make paths between nodes. An important aspect is that paths must not cross over other nodes, even though this may increase the length of a path by adding extra segments to it. Depending on how the paths are represented, edge routing may b:

- Shortest path routing: Nodes are connected only by straight which go directly from the source to the destination. In some implementations, it is acceptable for these lines to cross other nodes, while in others extra segments are added in order to go around obstacles.
- Arc routing: This approach represents paths as arcs. Nodes are usually placed in a linear layout (all nodes are placed on the same axis) and each edge is considered a diameter line for a circle. Then an arc of corresponding length is drawn to join the pair of nodes.
- Orthogonal routing: In orthogonal routing, paths are composed only of vertical and horizontal lines, joined by 90 degree bends. While it is not the most efficient method with regards to the space occupied by the drawing, it does provied clear and direct paths which do not cross over other elements.

3.3.1 Rules based genetic routing algorithm –adica implementarea mea, open for a better name–

Having presented the a range of algorithms which help achieve the goal of representing and drawing diagrams, we can now explain and exemplify how the implementation of this thesis works. We shall go through each of the three stepts: planarity testing, node placement and edge routing and detail the design of each level.

The proposed implementation starts by testing the planarity of the input graph using the criteria derived from Euler's theorem. These theorems are applied first because of their reduced complexity and because all the needed information is computed when the graph is first constructed. Should the result be negative, there is sufficient information to decide that the graph is not planar and that the drawing will have crossing edges. If, otherwise, the result is positive, we know that the theorems are only a necessary condition, not a sufficient one. At this point, the PQ-tree algorithm is applied to thoroughly decide if the graph is planar.

With the planarity property of the graph decided, the algorithm proceeds to the placing step. Here, the nodes will be assigned positions in the drawing space (the canvas) based on certain criteria. First of all, as seen in grid placing, the node with the highest numbed of connections is placed in the center of the canvas. Then, an are called the "restricted area" is computed around this central node. The central node's neighbours shall be placed in this area in a random order, and no other node may be placed in this area. This step ensures that the random factor is eliminated and a certain consistency is kept amongst consecutive drawings of the same graph. All the other nodes are placed randomly in the remaining area.

At this point, notions from genetic algorithms are introduced. The graph is considered a population and each node represents an individiual. For the remaining steps, each individual shall be evaluated by a fitness function and suffer modifications depending of the result. In order to change and evolve the population, the position of a node shall be "recombined". Each node outside the "restricted area" shall generate, along with its neighbour closest to the center, a new pair of nodes which are either closer to each other (attracted) or further apart from each other (rejected), as described by the force directed method. This step shall continue until a certain number of steps has been exhausted (we have explored the maximum allowed samples of populations) or the system has reached stability.

**** Sa pun poze care ilustreaza in cativa pasi ce balmajesc aici? ******

**** also pentru fitness folosesc niste registri care tin minte ce reguli fute un nod. pomenesc aici sau la architecture? ******

The last step is the edge routing. The method used is orthogonal edge routing, however the process starts as a shortest path routing. The initial paths are direct lines between two nodes. Then, the paths are checked to see if they do not cross any obstacles. Should it cross another node, the path is split into orthogonal segments which go around the obstacle. This step is repeated until the path is clear of all obstacles and is incidental only to the source and destination. Lastly, should the dawn graph be planar, the drawing is checked for edge intersection. Paths which do not respect the planarity property are recomputed.

Chapter 4

Software Platform

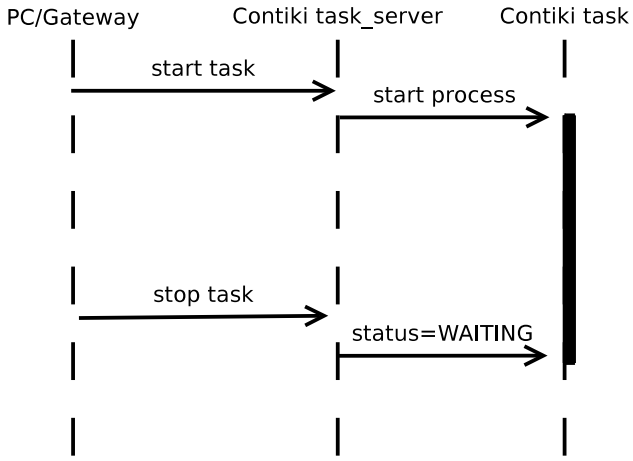

Figure 4.1: *The exchange of messages while starting/stopping tasks*

Listing 4.1: Task server snippet

```
1  PROCESS_THREAD(task_server_process, ev, data)
2  {
3      PROCESS_BEGIN();
4
5      list_init(task_list);
6
7      list_add(task_list,&el_monitor_process);
8      list_add(task_list,&el_delay_process);
9      list_add(task_list,&el_temperature_sensing);
10
```

```
11      tcp_listen(HTONS(1010));
12
13      while(1)
14      {
15          PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
16
17          if(uip_connected())
18          {
19              PSOCK_INIT(&ps, buffer, sizeof(buffer));
20
21              while(!(uip_aborted() || uip_closed()
22                  || uip_timedout()))
23              {
24                  PROCESS_WAIT_EVENT_UNTIL
25                        (ev == tcpip_event);
26                  handle_connection(&ps);
27              }
28          }
29      }
30      PROCESS_END();
31  }
```

Chapter 5

Other Chapters, TBD

Chapter 6

Conclusions and Future Work

Appendix A

Contiki API

A.1 Process macros

* `PROCESS_THREAD (name, ev, data )` - Define the body of a process. This macro is used to define the body (protothread) of a process. The process is called whenever an event occurs in the system, A process always starts with the PROCESS_BEGIN() macro and end with the PROCESS_END() macro.
* `PROCESS_BEGIN ()` - Define the beginning of a process.
* `PROCESS_END ()` - Define the end of a process.
* `PROCESS_YIELD ()` - Yields the currently running process
* `PROCESS_WAIT_EVENT_UNTIL (c)` - Wait for an event to be posted to the process, with an extra condition. This macro is very similar to PROCESS_WAIT_EVENT() in that it blocks the currently running process until the process receives an event. But PROCESS_WAIT_EVENT_UNTIL() takes an extra condition which must be true for the process to continue.
* `PROCESS_PAUSE` - Yield the process for a short while. This macro yields the currently running process for a short while, thus letting other processes run before the process continues.

A.2 uIP functions

* `PSOCK_INIT (psock, buffer, buffersize)` - Initializes a protosocket. This macro initializes a protosocket and must be called before the protosocket is used. The initialization also specifies the input buffer for the protosocket.

* `PSOCK_SEND (psock, data, datalen)` - Send data. This macro sends data over a protosocket. The protosocket protothread blocks until all data has been sent and is known to have been received by the remote end of the TCP connection.
* `PSOCK_READBUF (psock)` - Read data until the buffer is full. This macro will block waiting for data and read the data into the input buffer specified with the call to PSOCK_INIT(). Data is read until the buffer is full..
* `CCIF process_event_t tcpip_event` - The uIP event. This event is posted to a process whenever a uIP event has occurred.
* `CCIF void tcp_listen (u16_t port)` - Open a TCP port. This function opens a TCP port for listening. When a TCP connection request occurs for the port, the process will be sent a tcpip_event with the new connection request.
* `struct uip_conn *tcp_connect(uipipaddr_t *ripaddr,u16 port, void *appstate)` - This function opens a TCP connection to the specified port at the host specified with an IP address. Additionally, an opaque pointer can be attached to the connection. This pointer will be sent together with uIP events to the process.
* `uip_connected()` - Has the connection just been connected?
* `uip_closed()` - Has the connection been closed by the other end?
* `uip_aborted()` - Has the connection been aborted by the other end?
* `uip_timedout()` - Has the connection timed out?
* `uip_newdata()` - Is new incoming data available?
* `uip_close()` - Close the current connection.

Appendix B

Node capabilities

Task	AVR Raven™	Sparrow	Sparrow Power
Temperature sensing	✔	✔	
Humidity sensing		✔	
Voltage & Current sensing			✔
Event detection	✔	✔	✔
Alarm beep	✔		
LED signal	✔	✔	

Table B.1: Node capabilities

List of Figures

List of Tables

Listings